# Java New Features

# Java Modules

# Issues in legacy approach

- Classes of all packages are exposed and available.
- Only restriction possible is to declare classes with default access
- Application becomes bulky as all packages are shipped with the application (typically in small devices)
- There is no way to restrict visibility of packages (no private packages)

# Java Module System (JPMS)

- A Module is a group of closely related packages and resources along with a new module descriptor file
- The main aim of the system is to collect Java packages and code to be collected into a single unit called a Module
- it's a "package of Java Packages" abstraction that allows us to enhance code reusability
- Each module is responsible for its resources, like media or configuration files
- Java module is packaged as a *modular JAR file*.
- A Java module can specify which of its packages should be visible to other Java modules and which other Java modules are required for its own classes

# Java Module Basics

- A Java module must be given a unique name (similar to package name)
- A valid module name could be

  com.trg.module1

- Each Java module needs a Java module descriptor named module-info.java which has to be located in the module root directory

```
module com.trg.module1{
        // module entries
}
```

- To use a module, include the *jar* file into modulepath instead of the classpath
- module jar file added to classpath is normal jar file and module-info.class file will be ignored

# Module Descriptor

Descriptor includes several aspects of the module:

- **Name** – the name of our module
- **Dependencies** – a list of other modules that this module depends on
- **Public Packages** – a list of all packages we want accessible from outside the module
- **Services Offered** – we can provide service implementations that can be consumed by other modules
- **Services Consumed** – allows the current module to be a consumer of a service
- **Reflection Permissions** – explicitly allows other classes to use reflection to access the private members of a package

# Module Declarations

## exports

- By default, a module doesn't expose any of its API to other modules.
- This strong encapsulation was one of the key motivators for creating the module system in the first place.
- We use the exports directive to expose all public members of the named package:

```
module my.module {
    exports com.my.package.name;
}
```

# Module Declarations(contd)

## exports to

- We can restrict which modules have access to our APIs using the exports…to directive.
- We list which modules we are allowing to import this package as a requires using exports to

```
module my.module {
    exports com.my.package to other.mod1, other.mod2;
}
```

# Module Declarations(contd)

## requires

- This directive allows us to declare module dependencies:

```
module my.module {
    requires com.other;
}
```

- Now, my.module has both a runtime and a compile-time dependency on com.other.

- And all public types exported from a dependency are accessible by my.module

# Module Declarations(contd)

## requires static

- This directive specifies optional dependency:
- It is manadatory for the module to be available at compile time but optional at runtime

```
module my.module {
    requires static your.module;
}
```

- your.module should be present at compile time. But may not be present at runtime

# Module Declarations(contd)

## requires transitive

- This directive specifies transitive dependency:

```
module java.sql {
    requires transitive java.logging;
    requires transitive java.transaction.xa;
    exports java.sql;
    exports javax.sql;

}
```

- any module that reads *java.sql* (usually by requiring it) will automatically be able to read *java.logging*, *java.transaction.xa*

# Module Declarations(contd)

## open / opens

- Open keyword can be used to provide reflective access to any module
- An open module grants reflective access to all of its packages to other modules
- To provide reflective access to packages

```
module app.mod1{
    opens com.pkg1;
}
```

- To provide reflective access to the module

```
open module app.mod1{
}
```

# Module Declarations(contd)

- Circular dependency not allowed
- Compiling a module

  Javac -d out --module-source-path src/main/java  --module com.mymodule
- Running modules

  java --module-path out --module com.mymodule/com.jenkov.mymodule.Main

# Types of modules

- System Modules
- Application Modules
- Automatic Modules
- Unnamed Modules

# Types of modules(contd)

## System Modules

- These are the modules provided by JDK

- We can display these modules with

  java –list-modules

- java.base module is automatically available
- For other modules we need requires option

## Application Modules

- These are the modules which are application specific

# Types of modules(contd)

## Automatic Modules

- An automatic module is a jar that we put on the modulepath.
- There is a number of pre-existing libraries that are not modularized and can be used in our applications
- To facilitate migration, we can add any library's jar file to an application's module path, then use packages in that jar file.
- The jar's filename becomes its module name that must be a valid Java identifier that can be used in "requires" directive
- All packages of automatic modules are exported

# Types of modules(contd)

## Unnamed Modules

- From Java 9 all Java classes must be located in a module for the Java VM to use them
- On the **classpath** you can include all your older Java classes. All classes found on the classpath will be included in what Java calls the *unnamed module*
- The unnamed module *exports* all its packages
- However, the classes in the unnamed module are only readable by other classes in the unnamed module - or from automatic modules
- **No named module can read the classes of the unnamed module.**
- All classes in the unnamed module *requires* all modules found on the module path. That way, all classes in the unnamed module can read all classes exported by all the Java modules found on the module path

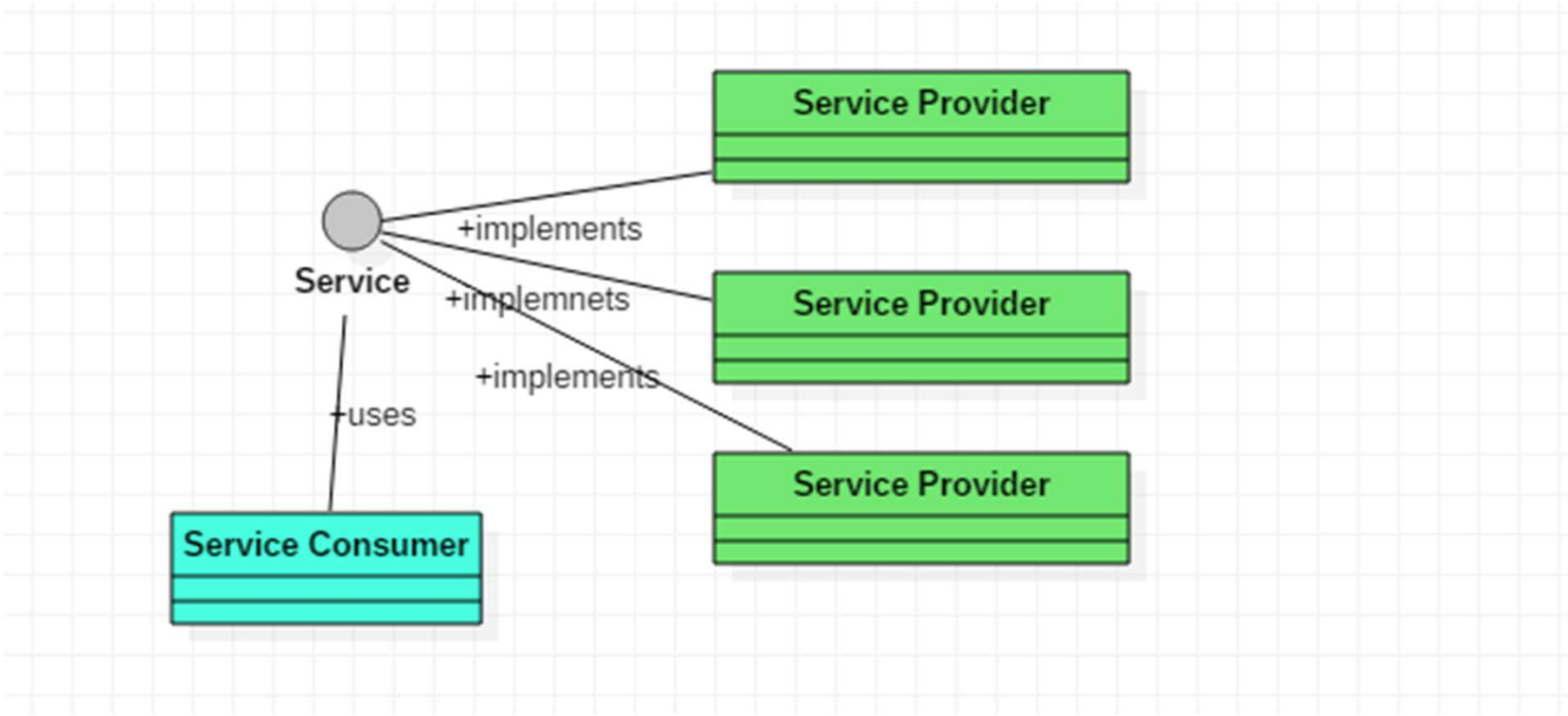
# Java Services

# Java Module Services

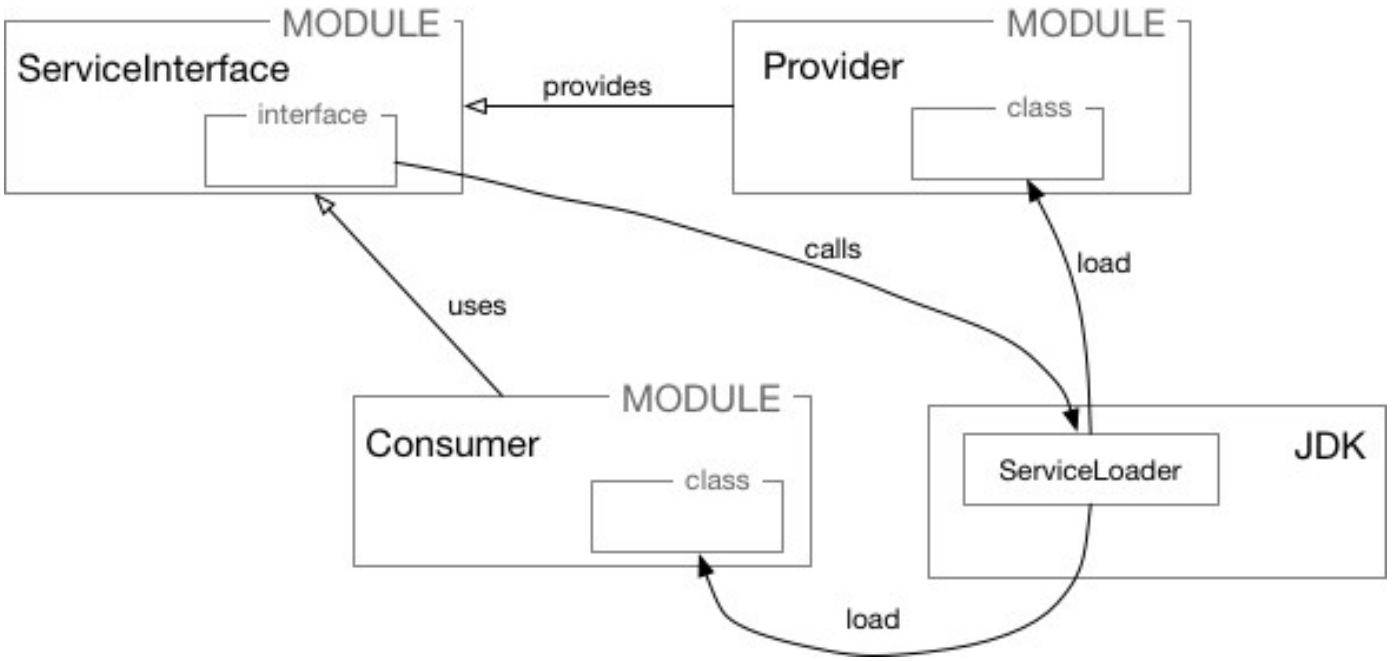
- A specific functionality provided by the library is known as service. .
- A service is defined by a set of interface and classes
- There are multiple implementations for a service and they are called as service providers.
- The client using this service will not have any contact with the implementations
- The ServiceLoader class (java.util) in JDK is responsible for discovering and loading all the service providers at a runtime for a service interface
- The ServiceLoader class allows decoupling between the providers and consumers
- The consumer knows only about the service interface

# Java Module Services

- This architecture follows plugin mechanism in which the service providers can be added or removed without affecting the service interface and the consumers



# Java Module Services



# Java Module Services

- **Service Interface**: It is an interface or abstract class that a Service defines
- **Service Provider**: Has specific implementations of a service interface. A Service could have zero, one, or many service providers
- **ServiceConsumer**: The class tagged to interface and and get a provider with the help of ServiceLoader

- **ServiceLoader**: The main class used to discover and load a service implementation

# Declarations

- **Service Interface**:

```
module ServiceInterface {
  exports com.msg;
}
```

# Declarations

- **Service Provider**:
- Create classes implementing the service interface

```
module mod.provider1 {
   requires mod.service;


   provides com.service.MessageService
        with com.provider1.MsgProvider1;
}
```

# Declarations

- **Service Consumer**:
- Create consumer class

```
module mod.consumer {
    requires mod.service;
    uses com.msg.MessageService;
```

# ServiceLoader

- ServiceLoader is a generic class in java.util package.
- Service providers are loaded by the load( ) method.

```
public static <S> ServiceLoader<S> load(Class <S> serviceType)
```

- ‘S’ specifies the service type which specifies the Class object for the desired service type.
- ServiceLoader.load(serviceType) returns an instance of ServiceLoader, which itself implements Iterable. So we can iterate over the result.
- ServiceLoader finds all the providers based on the module declarations

# Sealed Classes

# Sealed Classes

- The release of Java SE 17 introduces sealed classes
- This feature is about enabling more fine-grained inheritance control in Java. Sealing allows classes and interfaces to define their permitted subtypes.
- In other words, a class or an interface can now define which classes can implement or extend it.
- It is a useful feature for domain modeling and increasing the security of libraries

# Why Sealed Classes

- As an example, imagine a business domain that only works with cars and trucks, not motorcycles
- When creating the *Vehicle* abstract class, we should be able to allow only *Car* and *Truck* classes to extend it
- In that way, we want to ensure that there will be no misuse of the *Vehicle* abstract class within our domain
- we are more interested in the clarity of code handling known subclasses than defending against all unknown subclasses

# Declaring Sealed Classes

- To seal a class, add the **sealed** modifier to its declaration.
- Then, after any extends and implements clauses, add the **permits** clause.
- This clause specifies the classes that may extend the sealed class

```
public sealed class Shape permits Circle, Square, Rectangle
{

 }
```

# Subclasses of Sealed Class

- All subclasses should have any of these modifiers
  - **sealed:** It can only be extended by its permitted subclasses
  - **non-sealed:** It can be extended by unknown subclasses
  - **final:** subclass cannot be extended further.

# Constraints on Subclasses

- They must be accessible by the sealed class at compile time
- They must directly extend the sealed class

- They must have exactly one of the following modifiers to describe how it continues the sealing initiated by its superclass:
  - final: Cannot be extended further
  - sealed: Can only be extended by its permitted subclasses
  - non-sealed: Can be extended by unknown subclasses; a sealed class cannot prevent its permitted subclasses from doing this

- They must be in the same module as the sealed class (if the sealed class is in a named module) or in the same package (if the sealed class is in the unnamed module).

# Sealed Interfaces

- To seal an interface, we can apply the *sealed* modifier to its declaration.
- The *permits* clause then specifies the classes that are permitted to implement the sealed interface:

```
public sealed interface Service permits Car, Truck {

    int getMaxServiceIntervalInMonths();

    default int getMaxDistanceBetweenServicesInKilometers() {
        return 100000;
    }

}
```

# Classes in same file

- By defining permitted subclasses in the same file as the sealed class, we can omit permits clause

```
public sealed class Figure
{ }

final class Circle extends Figure {
    float radius;
}
non-sealed class Square extends Figure {
    float side;
}
sealed class Rectangle extends Figure {
    float length, width;
}
```

# Records as sub classes

- We can name a record class in the permits clause of a sealed class or interface
- Record classes are implicitly final, so modifier required

```
sealed interface Expr permits ConstantExpr, PlusExpr {
    public int eval();
}

record ConstantExpr(int i) implements Expr {
    public int eval() { return i(); }
}

record PlusExpr(Expr a, Expr b) implements Expr {
    public int eval() { return a.eval() + b.eval(); }
}
```

# Reflection for sealed classes

- Sealed classes are also supported by the reflection API
- The isSealed() returns boolean
- getPermittedSubclasses() returns an array of objects representing all the permitted subclasses
- Samples:

```
truck.getClass().isSealed()
truck.getClass().getSuperclass().isSealed()
truck.getClass().getSuperclass().getPermittedSubclasses()
```

# Records

# Concept of immutable classes

- Commonly, we need classes to simply hold data, such as database results, query results etc
- In many cases, this data is immutable
- To accomplish this, we create data classes with lot of boiler plate code
    1. *private*, *final* field for each piece of data
    2. getter for each field
    3. *public* constructor with a corresponding argument for each field
    4. *equals* method that returns *true* for objects of the same class when all fields match
    5. *hashCode* method that returns the same value when all fields match
    6. *toString* method that includes the name of the class and the name of each field and its corresponding value

# Records in Java

- Records are immutable data classes that require only the type and name of fields.
- The equals, hashCode, and toString methods, as well as the private, final fields and public constructor, are generated by the Java compiler.
- To create a Person record, we'll use the record keyword:

```
public record Person (String name, String address) {}
```

# What is generated

- Constructor

```
public Person(String name, String address) {
    this.name = name;
    this.address = address;
}
```

- public getters methods, whose names match the name of our field, for free.

```
public String name(){
}
public String address(){
}
```

# What is generated (contd)

- equals() method which returns true if the supplied object is of the same type and the values of all of its fields match
- *hashCode()* method which returns the same value for two objects if all of the field values for both objects match
- toString() method that returns a string containing the name of the record, followed by the name of each field and its corresponding value in square brackets.

  Person[name=John Doe, address=100 Linda Ln.]

# Customization

- we can customize our constructor implementation. This is called compact constructor

- This customization is intended to be used for validation and should be kept as simple as possible.

- For example, we can check for null values

```
public record Person(String name, String address) {
    public Person {
        Objects.requireNonNull(name);
        Objects.requireNonNull(address);
    }
}
```

# Customization(contd)

- We can also create new constructors with different arguments by supplying a different argument list:

```
public record Person(String name, String address) {
    public Person(String name) {
        this(name, "Unknown");
    }
}
```

# Customization(contd)

- creating a constructor with the same arguments as the generated constructor is valid, but this requires that each field be manually initialized:

```
public record Person(String name, String address) {
    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }
}
```

# Customization(contd)

- Additionally, declaring a compact constructor and one with an argument list matching the generated constructor leads to compilation error.
- the following is not valid

```
public record Person(String name, String address) {
    public Person {
        Objects.requireNonNull(name);
        Objects.requireNonNull(address);
    }

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }
}
```

# Customization(contd)

- Additionally, we can add the following
  - static fields
  - static methods
  - Any other methods
  - Nested records

- We can also create generic records

```
record Container<T>(int id, T value) {  }
Container<Integer> intContainer = new Container<>(1, Integer.valueOf(1));
Container<String> stringContainer = new Container<>(1, "1");

Integer intValue = intContainer.value();
String strValue = stringContainer.value();
```

# switch expressions

# switch expressions

- The traditional switch statement is procedural.
- It lacks the capability to return a value directly.
- The new switch expression is an expression rather than a statement, which means it evaluates to a value
- the switch expression uses a syntax similar to arrow functions to map case clauses directly to values
- Arrow format allowed in switch statement also
- Unlike switch statement, switch expression needs default as it has to evaluate to a value
- Format:

case *label1*, *label2*, ..., *labelN* ->

*expression*;|*throw-statement*;|*block*

# switch expression – example1

```
public enum Day { SUNDAY, MONDAY, TUESDAY,WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY};

    Day day = Day.WEDNESDAY;
    int dayNum =
        switch (day) {
            case MONDAY, FRIDAY, SUNDAY -> 6;
            case TUESDAY                -> 7;
            case THURSDAY, SATURDAY     -> 8;
            case WEDNESDAY              -> 9;
            default -> throw new IllegalStateException("Invalid day: " + day);
        }
```

# switch expression – example2

```
int days=0;
int month = 2;
days = switch (month) {
    case 1,3,5,7,8,10,12 -> 31;
    case 4,6,9,11 -> 30;
    case 2 -> 28;
    default -> throw new IllegalStateException("Invalid month: "
+ month);
};

System.out.println(days);
```

# yield statement

- For a case, statement block makes it switch statement
- yield statement makes it expression
- yield will  return a value

# switch expression – example2

```
Day day = Day.WEDNESDAY;
    int numLetters = switch (day) {
        case MONDAY:
        case FRIDAY:
        case SUNDAY:
            System.out.println(6);
            yield 6;
        case TUESDAY:
            System.out.println(7);
            yield 7;
        case THURSDAY:
        case SATURDAY:
            System.out.println(8);
            yield 8;
        case WEDNESDAY:
            System.out.println(9);
            yield 9;
        default:
            throw new IllegalStateException("Invalid day: " + day);
    };
```

# switch expression – example3

With case yield both : and -> are allowed but different format
With : all statements can provided as it is
With -> all statements should be part of a block

```
int days = 0;
int month = 7;
// using arrow format of java 12
days = switch (month) {
case 1, 3, 5, 7, 8, 10, 12 -> {
System.out.println("31 days month");
yield 31;
}
case 4, 6, 9, 11 -> {
System.out.println("30 days month");
yield 30;
}
case 2 -> {
System.out.println("February");
yield 28;
}

default -> throw new IllegalStateException("Invalid month: " + month);
};
```

# Pattern Matching

# Pattern Matching

- Pattern matching involves testing whether an object has a particular structure, then extracting data from that object if there's a match
- Basically, this is required in polymorphism where we do not know the type of present object
- Pattern matching introduces new language enhancements

# Pattern Matching

- Applying type patterns to the instanceof operator simplifies type checking and casting. Moreover, it enables us to combine both into a single expression:

```
if (o instanceof String s) {
    System.out.printf("Object is a string %s", s);
} else if (o instanceof Number n) {
    System.out.printf("Object is a number %n", n);
}
```

- This built-in language enhancement helps us write less code with enhanced readability.

# Pattern Matching

- With Java 17, the application of pattern matching now also expands to **switch** expressions
- However, it is still a preview feature, so we need to enable preview to use it:

- java --enable-preview --source 17 PatternMatching.java

```
static double getDoubleUsingSwitch(Object o) {
    return switch (o) {
        case Integer i -> i.doubleValue();
        case Float f -> f.doubleValue();
        case String s -> Double.parseDouble(s);
        default -> 0d;
    };
}
```

# Misc

# Text Blocks

- Since Java 15, text blocks are available as a standard feature
- Text Blocks allow to create multi line string
- Text blocks start with a “”” (three double-quote marks)
- The most simple example looks like this

```
String str= """
   <html>
     <body>
        <span>example text</span>
     </body>
  </html>""";
```

# Escaping double qoutes

- Inside text blocks, double-quotes don’t have to be escaped.
- We could even use three double-quotes again in our text block by escaping one of them:

```
public String getTextWithEscapes() {
    return """
           "fun" with
           whitespace
           and other escapes \"""
           """;
}
```

# Escaping Line Terminators

- we might have long lines of text in our source code that we want to format in a readable way
- We can escape a newline so that it is ignored:

```
String str =  """
        This is a long test which looks to \
        have a newline but actually does not""";
```

- this String literal will just equal a normal non-interrupted String:

# Trailing Spaces

- The compiler ignores all trailing spaces in text blocks.
- However, we can escape a space using the new escape sequence \s.
- The compiler will also preserve any spaces in front of this escaped space.

```
String str =
    """
        line 1
        line 2      \s
        """;
```

# String new methods

| | |
|---|---|
| **lines()** | Returns a steam of strings split at newline |
| **repeat(int)** | Repeats the string concatenated |
| **formatted(Objectt… args)** | Returns formatted string |
| **isBlank()** | Returns true if string is empty or contains only white space |
| **indent(int)** | adjusts the indentation of each line of this string with the specified number of spaces |
| **strip()**<br>**stripTrailing()**<br>**stripLeading()** | Strips spaces. trim() works for ASCII<br>strip() works for ASCII and UNICODE |

# readString(), writeString()

- The writeString() method of Files Class in Java is used to write contents to the specified file.

  Files.writeString(path, string, options)

- options – Different options to enter the string in the file. Like append etc

- The readString() method of Files class reads a string

# var keyword

- The **var reserved type name (not a Java keyword)** is used for type inference
- var keyword detects automatically the datatype of a variable based on the surrounding context
- var cannot be used for generic types
- Examples:

```
var x = 100; // int
var y = 1.90;   // double
var z = 'a';   // char
var p = "tanu";  // String
var q = false;    // boolean
var<Integer>   vi  = new ArrayList<Integer>() ;   // not allowed
```

# HttpClient

# Legacy HTTP Client operations

- *HttpUrlConnection* class has been present in the JDK since the very early years of Java.
- *HttpURLConnection* API is low-level and isn't known for being feature-rich and user-friendly
- URLConnection API was designed with multiple protocols that are now no longer functioning (FTP, gopher, etc.)
- The API predates HTTP/1.1 and is too abstract
- It works in blocking mode only (synchronous)
- It is very hard to maintain

# HTTPClient

- HTTP Client provides synchronous and asynchronous request mechanisms

- The API consists of three core classes:
  - ***HttpClient*** behaves as a container for configuration information common to multiple requests.
  - ***HttpRequest*** represents the request to be sent via the *HttpClient*.
  - ***HttpResponse*** represents the result of an *HttpRequest* call

# HTTPRequest

- *HttpRequest* is an object that represents the request we want to send
- New instances can be created using *HttpRequest.Builder*
- We can get it by calling *HttpRequest.newBuilder()*.
- *Builder* class provides a bunch of methods that we can use to configure our request
- Some of the methods:

```
uri(new URI("http://hello.com"))
GET()
POST(BodyPublisher body)
PUT(BodyPublisher body)
DELETE()
headers("key1", "value1", "key2", "value2")
```

# HTTPRequest(contd)

- We can add a body to a request by using the request builder methods: *POST(BodyPublisher body)*, *PUT(BodyPublisher body)* and *DELETE()*.
- The API provides a number of *BodyPublisher* implementations :

  *StringProcessor* – reads body from a *String*, created with *HttpRequest.BodyPublishers.ofString*

  *InputStreamProcessor* – reads body from an *InputStream*, created with *HttpRequest.BodyPublishers.ofInputStream*

  *ByteArrayProcessor* – reads body from a byte array, created with *HttpRequest.BodyPublishers.ofByteArray*

  *FileProcessor* – reads body from a file at the given path, created with *HttpRequest.BodyPublishers.ofFile*

*For sample Code refer: https://www.baeldung.com/java-9-http-client*

# HTTPClient

- All requests are sent using *HttpClient*, which can be instantiated using the *HttpClient.newBuilder()* method or by calling *HttpClient.newHttpClient()*.
- It provides a lot of useful and self-describing methods we can use to handle our request/response.

- Ex:

```
HttpClient client = HttpClient.newHttpClient();
HttpResponse<String> response = client.send(request,
BodyHandlers.ofString());
```

# HTTPResponse

- The *HttpResponse* class represents the response from the server. It provides a number of useful methods
- *statusCode()* returns status code (type *int*) for a response
- *body()* returns a body for a response (return type depends on the response *BodyHandler* parameter passed to the *send()* method)

# async requests

- Basic send() method of HttpClient is blocking method
- Application blocks till response received from server
- Now we can also send async request which is non blocking
- The asynchronous API returns immediately with a **CompletableFuture** that completes with the HttpResponse when it becomes available

```
client.sendAsync(request, BodyHandlers.ofString())
      .thenApply(response -> {
                              System.out.println(response.statusCode());
                               return response; } )
      .thenApply(HttpResponse::body)
      .thenAccept(System.out::println);
```